Un Análisis preliminar sobre reparación de modelos *Alloy* utilizando *Sketching*

César Cornejo[1,2], Germán Regis[1], and Nazareno Aguirre[1,2]

[1] Departamento de Computación, FCEFQyN,
Universidad Nacional de Rio Cuarto,
{ccornejo, gregis, naguirre}@dc.exa.unrc.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Resumen El tamaño y complejidad de los sistemas de software modernos muestran de manera taxativa la necesidad e importancia de contemplar las etapas tempranas en el desarrollo de software. En particular, una de estas etapas que permite tener una comprensión más abstracta y general del sistema como un todo, es la etapa de *modelado*. Si bien existe una variada gama de lenguajes para tal fin, dos características que consideramos importantes para su elección son su *output* como entrada para las etapas siguientes y su versatilidad en el análisis. En este sentido, los lenguajes con algún grado de formalismo subyacente prevalecen al permitir construir herramientas automáticas o semi-automáticas para su procesamiento.
Al igual que en las siguientes etapas del desarrollo, el *modelado* no se encuentra exento de errores como producto de una actividad humana. Para abordar este problema, diferentes técnicas y herramientas fueron propuestos. En este trabajo proponemos combinar dos técnicas conocidas con el objetivo de reparar posibles errores en modelos especificados en Alloy. Utilizando el *testing* como herramienta para localizar errores, nuestra técnica emplea el concepto de *Sketching* para descubrir y proponer una posible reparación de los mismos.

1. Introducción.

Debido al tamaño y complejidad de los sistemas de software modernos, la utilización de *modelos* en etapas tempranas del desarrollo es una técnica común, principalmente porque nos permite comprender y razonar desde un punto de vista más abstracto, alejados de los detalles de la implementación.

Está comprobado que esta comprensión abstracta del sistema contribuye a disminuir posibles errores en etapas futuras del desarrollo. Aún así, la etapa de modelado no deja de ser una actividad realizada por humanos y en consecuencia, no está exenta de errores. En este sentido, contar con la posibilidad de razonar formalmente, con sustento matemático, sobre estos modelos es una característica importante a la hora de escoger lenguajes para realizar esta tarea. Por esta razón, y cada vez con mayor frecuencia, es común encontrar algún grado de formalismo y sistematización en etapas tempranas de desarrollo como por ejemplo, captura de requisitos [12,1,2] además de modelado.

En general, el grado de popularidad de los lenguajes, en cualquier etapa del desarrollo de software, está fuertemente influenciado por las herramientas y recursos asociados a él. Si bien esta observación es evidente en los lenguajes de programación, durante la implementación, no es menos importante en las etapas tempranas . Contar con técnicas y herramientas automáticas o semi-automática posibilita entre otras cosas, analizar y verificar de manera sistemática ciertas propiedades deseables de nuestros modelos y actuar en consecuencia.

En este sentido, numerosas técnicas de análisis y verificación han sido propuestas para la etapa de modelado, como por ejemplo, la noción de *modelchecking* basado en representación explícita de estados y utilizando lógicas modales para la especificación de propiedades [7] o técnicas basadas en *sat-solving* que exploran de manera exhaustiva los posibles comportamientos de modelos acotados [6,10]. Más aún algunas técnicas con gran difusión y uso en etapas avanzadas, como el *testing*, se han extrapolado a las etapas iniciales para lograr una mayor confianza los modelos. Un ejemplo de ello es AUnit [11], una herramienta que permite generar y analizar casos de prueba unitario (test de unidad) para modelos Alloy.

Este trabajo se situa en el escenario donde tenemos un modelo Alloy como producto inicial para nuestro sistema y luego de analizarlos mediante pruebas unitarias encontramos errores, es decir, ciertos casos de prueba *no* son satisfechos por nuestro modelo. En esta situación, aprovechando la formalidad subyacente y utilizando mecanismos automáticos, proponemos una técnica que permita reparar automáticamente el modelo defectuoso en cuestión. Para lograr tal fin, utilizando la información de los casos de prueba no superados y la técnica de *sketching* para generar automáticamente predicados que, al sustituir a los originales, permitan alcanzar el caso de prueba fallidos. Si bien, las técnicas basadas en *testing* no garantizan una confianza absoluta [3,15], creemos que aportan confianza en nuestro trabajo y son plausiblesde aplicar en la práctica gracias a su escalabilidad.

El trabajo procede de la siguiente manera: inicialmente repasaremos los conceptos subyacentes, mostrando de manera sintética el lenguaje elegido y el funcionamiento de las herramientas asociadas. Luego en la sección 3, explicaremos nuestra técnica, enfocándonos en *sketching* como técnica de reparación. Finalmente en las secciones 4 y 5 mencionaremos el rumbo a seguir y expondremos las conclusiones.

2. Preliminares

En esta sección realizaremos una breve revisión de los lenguajes y herramientas subyacentes a nuestro trabajo. Como lo mencionamos anteriormente, el contexto de nuestro trabajo se sitúa en la etapa de modelado y particularmente estamos interesados en lenguajes de modelado con fundamentos formales.

Alloy [6] [5] es el lenguaje elegido para este trabajo, dado que es un lenguaje de especificación formal orientado a modelos. Esta definido en términos de una

semántica relacional, su sintaxis incluye constructores con notación orientada a objetos y soporta análisis automatizado acotado. Para ello provee su herramienta asociada Alloy Analyzer.

Un modelo Alloy es un conjunto de restricciones que describe implícitamente un conjunto de estructuras, estos modelos se definen esencialmente en términos de dominios de datos y operaciones entre estos dominios.

Para definir el dominio de datos utilizamos *signaturas*, para las operaciones sobre estos dominios el lenguaje nos provee *predicados* y *funciones*. Además nos permite definir restricciones sobre el modelo con fórmulas que se imponen como válidas, es decir *axiomas*. Una vez especificado nuestro modelo, mediante la utilización del Alloy Analyzer podemos buscar instancias que lo satisfagan o bien, mediante la definición de *aserciones*, podemos explorar exhaustivamente todas las posibles instancias (acotadas) del modelo en busca de contraejemplos que las contradigan.

La Fig. 1 presenta un ejemplo en el que definimos *listas simplemente encadenadas*, nuestro dominio en este modelo son las signaturas *List* y *Node*, para completar la especificación necesitamos agregar la propiedad de aciclicidad (*Acyclic*).

```
1  one sig List {
2    header: lone Node
3  }
4
5  sig Node {
6    link: lone Node
7  }
8
9  pred Acyclic() {
10  all n: Node | n in List.header.*link => n !in n.^link
11 }
```

Figura 1. Ejemplo de modelo *Alloy*

Una vez que tenemos nuestro modelo especificado, una pregunta que surge es si expresa exactamente lo que esperamos. Una forma de ganar confianza aplicando técnicas más simples que la especificación de propiedades, es realizar *testing*. Si bien no da garantías absolutas, poder llevar a cabo esta tarea sólo requiere de conocer instancias válidas (concretas) que nuestro modelo debería cumplir.

AUnit [11] es una herramienta que extiende al Alloy Analyzer que nos permite escribir y ejecutar casos de prueba (tests) sobre nuestra especificación Alloy. Esta extensión nos brinda un entorno de prueba formal y presenta una gramática adicional para escribir los casos de prueba. Continuando con el ejemplo de listas simplemente encadenadas, la Figura 2 muestra un ejemplo de uso de AUnit.

Además, la herramienta admite la generación automática de casos de prueba que puede aplicarse sobre todo nuestro modelo, o bien parcialmente mediante la selección de predicados particulares. La herramienta soporta una interfaz fácil

de utilizar y con reportes avanzados como por ejemplo métricas de cobertura de código.

```
val Test5{
        some disj List0 : List | some disj Node0, Node1 : Node |  {
                List = List0
                Node = Node0 + Node1
                header = List0->Node0
                link = Node0->Node0
                @cmd:{ !Acyclic[] } }
}
@Test Test5: run Test5 for 3 expect 1
```

Figura 2. Ejemplo de AUnit test

En ciertas ocasiones, la etapa de modelado, puede ser una sucesión de refinamientos sobre un modelo parcial inicial. Para abordar estos casos, no sólo es imprescindible contar con un lenguaje que nos de soporte, sino también contar con técnicas que ayuden a transitar los ciclos de refinamiento hasta llegar al modelo deseado. Ejemplos de ello son los Sistemas Transición Modales (MTS) [4] que permiten capturar diferentes nociones de obligación en los modelos y luego, de manera automática, sintetizar controladores que garanticen el cumplimiento de los objetivos. En el caso de Alloy, una técnica que podemos aplicar para "completar" modelos parciales es *sketching*.

ASketch [14] es una herramienta que dado un modelo Alloy con *holes* (huecos marcados sintacticamente) nos permite generar predicados de manera que el modelo satisfaga un conjunto de casos de prueba asociado a él. Estos *holes* o comodines pueden ser operadores binarios, de comparación, lógicos, cuantificadores, operadores unarios o incluso expresiones de navegación sobre el dominio del modelo como se detalla en la Tabla1. Estos operadores se anotan dentro de los predicados para debilitar (especificar parcialmente) restricciones o propiedades del modelo.

Como ejemplo, en la Figura3 se anota el predicado `Acyclic` para indicar la cuantificación sobre elementos de la signatura *Node*. Tal que, los elementos de node que cumplen con una comparación con otra expresión de navegación, necesariamente cumplen otra compración con una expresión de navegación que puede o no ser la misma.

Una vez anotado el modelo, ASketch genera los predicados candidatos, aquellos que cumplen con los *holes* indicados sintacticamente, pero que superen los casos de prueba asociado. El principal objetivo de esta herramienta es ayudar al usuario a especificar un modelo Alloy de manera correcta en base a las instancias que describen los casos de prueba asociado.

```
pred Acyclic() {
 -- holes:
 \Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\
 }
```

Figura 3. Predicado *Acyclic* con *holes*

Hole	Notación	Candidatos
Operador binario	\BO\	&, +, -
Operador de comparación	\CO\	=, in, !=, !in
Operador lógico	\LO\	||, &&, <=>, =>
Cuantificador	\Q\	all, no, some, lone, one
Operador Unario	\UO\	no, some, lone, one
Operador de expresión unaria	\UOE\	~, *, ^
Operador de formula unaria	\UOF\	!, ε
Expresión	\E\	cualquier expresión

Tabla 1. Operadores para escribir *holes* de ASketch.

3. Hacia la reparación de modelos Alloy con ASketch

Para evaluar *sketching* como técnica de reparación, partimos de un modelo correcto y generamos automáticamente una conjunto de casos de prueba para el mismo mediante AUnit. Luego utilizaremos este conjunto para definir si las reparaciones candidatas verdaderamente lo son.

En pos de generar un escenario para evaluar nuestra propuesta, incorporamos al modelo una serie de errores. Cabe destacar que comprobamos que dichos errores sean detectados por los casos de prueba que generamos anteriormente.

Aprovechando el análisis que realiza AUnit sobre nuestro modelo erróneo, y en particular el reporte de cobertura de código, podemos distinguir cuáles son los predicados que infringen a los casos de prueba no superados. Una ves identificados, procedemos a anotar dichos predicados con los operadores (*holes*) que mencionamos anteriormente en la Tabla 1, es decir, debilitamos la especificación de dicho predicado, supuestamente erróneo, tornándolo parcial.

Ahora sí, tomando esta especificación parcial comenzamos el proceso de generación mediante ASketch, una vez obtenidos todos los predicados candidatos, los incorporamos (de a uno por vez en caso de haber más de uno) en el modelo erróneo sustituyendo al predicado parcial y contrastamos con todo el conjunto de casos de prueba. Cabe recordar que pueden existir más de un predicado potencialmente erróneo. Si logramos reemplazar todos los predicados erróneos, entonces hemos encontrado una *reparación* de nuestro modelo que pasa todos los casos de prueba asociados a él.

3.1. Casos de estudio

Para llevar adelante la evaluación que planteamos, tomamos de la literatura[14,8] distintos casos de estudio con su conjunto de casos de prueba que utilizaremos evaluar nuestra propuesta.

Single Linked List El modelo correcto para este caso es el utilizado como ejemplo en la sección anterior. En la Figura 5 mostramos el predicado *acyclic* con *holes* y tres candidatos generados por ASketch.

```
pred Acyclic() {
 -- holes:
 \Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\
 -- candidato 1:
 all n: Node | n in n.~^link => n !in List.header.^link
 -- candidato 2:
 all n: Node | n in n.^link => n !in List.header.^link
 -- candidato 3:
 all n : Node | n in List.header.^link => n !in n.~^link
}
```

Figura 4. Predicado Alloy con `holes`

Luego de probar los tres casos candidatos en AUnit, observamos que todos los casos pasan correctamente todos los casos de prueba que utilizamos para evaluar como reparación.

Grade Para este caso también generamos los casos de prueba automáticamente para el modelo. Además se modifica el predicado *PolicyAllowsGrading* insertando errores para lograr un modelo erróneo. La Figura 6 muestra el predicado con *holes* en donde AUnit identifica posibles errores y, finalmente, las soluciones generadas que propone ASketch.

En este caso de estudio, los tres predicados candidatos *no logran* pasar los casos de prueba, es decir, no logramos una reparación.

Árboles Coloreados Este ejemplo aborda la especificación de la conocida estructura de datos de árboles coloreados. La Figura 7 exhibe un modelo correcto para el mismo. De manera similar a los casos anteriores, se genera el conjunto de casos de prueba que oficiará de oráculo. Para conseguir un escenario más complejo, para este modelo vamos a considerar más de un predicado erróneo mediante su manipulación. Luego de ejecutar ASketch, evaluamos los candidatos consiguiendo las tres reparaciones.

4. Conclusiones

Alloy es un lenguaje de especificación formal muy utilizado para el modelado de sistemas, su herramienta Alloy Analyzer facilita un análisis completamente

```
abstract sig Person {}

sig Student extends Person {}

sig Professor extends Person {}

sig Class {
  assistant_for: set Student,
  instructor_of: one Professor
}

sig Assignment {
  associated_with: one Class,
  assigned_to: some Student
}

pred PolicyAllowsGrading(s: Person, a: Assignment) {
   s in a.associated_with.assistant_for || s in a.associated_with.instructor_of
   s !in a.assigned_to
}
```

Figura 5. Predicado Alloy Grades

```
pred PolicyAllowsGrading(s: Person, a: Assignment) {
  -- Holes:
  \E,e\ \CO,co\ \E,e\ \LO,lo\ \E,e\ \CO,co\ \E,e\
  \E,e\ \CO,co\ \E,e\
  -- Candidato 1:
  Class.assistant_for !in Class.assistant_for &&
  a.associated_with.assistant_for = a.associated_with.instructor_of
  Class.assistant_for in s
  -- Candidato 2:
  s != Class.instructor_of =>
  a.associated_with.assistant_for = a.associated_with.instructor_of
  a.assigned_to !in a.associated_with.assistant_for
  -- Candidato 3:
  a.associated_with.assistant_for !in a.associated_with.instructor_of =>
  a.assigned_to != s
  a.associated_with.assistant_for != Class.assistant_for
}
```

Figura 6. Predicado PoliciAllowsGrading *Hole* y candidatos de reparación

automatizado para explorar los modelos que genera. Las extensión que estudiamos son de muchas utilidad en ambos casos y nutren al análisis que se realiza en este tipo de lenguaje haciéndolo más completo y más simple para el usuario, tanto para evaluar nuestros modelos mediante técnicas como el *testing*, como así también para sintetizar predicados de especificaciones parciales, guiado por casos de prueba.

En este sentido de analizar y explorar de manera automática un modelo, analizar la reparación de los mismos de manera automática se acerca a completar las herramientas que facilitan el uso de este tipo de lenguajes y ayudan al usuario en su especificación.

Luego de la evaluación que realizamos y los resultados obtenidos, podemos concluir que la reparación de modelos mediante la técnica de *sketching* tiene

```
abstract sig Color {}
one sig Red extends Color {}
one sig Blue extends Color {}

sig Node {
  neighbors: set Node,
  color: one Color
}

pred undirected {
   neighbors = ~neighbors
   no iden & neighbors
}

pred graphIsConnected {
  all n1: Node | all n2: Node-n1 |
     n1 in n2.^neighbors
}

pred treeAcyclic {
  all n1, n2: Node |
     n1 in n2.neighbors => n1 !in n2.^(neighbors-(n2->n1))
}

run {} for 3 Node
```

Figura 7. Especificación Alloy para Árboles Coloreados

muchos puntos de interés para trabajar, es decir, tiene muchas potenciales optimizaciones a realizar. Cabe notar que esta técnica, guiada por casos de prueba no está exenta del problema abordado en [15]. Es decir, consideramos la reparación tomando como oráculo un conjunto de casos de prueba, lo cual, las garantías de correctitud están directamente determinadas por la calidad de dicho conjunto. En este sentido, al observar los resultados obtenidos, y en particular para el caso de estudio *Grade*, ASketch logra sintetizar predicados candidatos como reparación, pero al ser evaluados por la suite de test de aceptación los mismos no se comportan como un modelo correcto.

5. Trabajo futuro.

Tal como observamos en esta evaluación y en [14], la calidad de generación de predicados que *ASketch* nos provee esta ligada fuertemente a los conjuntos de casos de prueba y los fragmentos que utilizamos. Sobre los conjuntos de casos de prurba, el uso de herramientas automáticas de generación de test es una buena opción. *AUnit*nos permite generar test para los modelos de manera *bounded exhaustive* [11] ó realizando *mutation testing* [13]. Además de utilizar estos test generados de manera automática y reemplazar los propuestos por el usuario, también se puede mejorar la generación de test utilizando técnicas de *field exhaustive* [9]. Mediante esta técnica también pueden generarse de manera automática los fragmentos que utilizamos para realizar `sketching`. Ambos casos nos permitirían tener mejores recursos para la generación de predicados de *ASketch*, y así mejorar la calidad de los predicados generados.

```
pred undirected {
  -- Holes:
  \E,e1\ \CO,co\ \E,e1\
  \UO,uo\ \E,e1\
  -- Candidato 1:
    neighbors = ~neighbors
    no iden & neighbors
  -- Candidato 2:
    ~neighbors, =, neighbors
    no iden & neighbors
  -- Candidato 3:
    neighbors in ~neighbors
    no iden & neighbors
}

pred graphIsConnected {
  all n1: Node | all n2: Node-n1 |
    -- Holes:
    \E,e2\ \CO,co\ \E,e2\
    -- Candidato 1:
    n2.(neighbors) != n2.^(neighbors-n2->n1)
    -- Candidato 2:
    n2.^(neighbors-n2->n1) != n2.(neighbors)
    -- Candidato 3:
    n2.^(neighbors-n2->n1) != n2.(neighbors)
}

pred treeAcyclic {
  all n1, n2: Node |
    -- Holes:
    \E,e2\ \CO,co\ \E,e2\ \LO,lo\ \E,e2\ \CO,co\ \E,e2\
    -- Candidato 1:
    n2.^(neighbors-n2->n1) in n2.(neighbors) <=>
    n2.^(neighbors-n2->n1) in n2.(neighbors-n2->n1)
    -- Candidato 2:
    n1 !in n2.(neighbors) <=> n2.(neighbors) in n2.^(neighbors-n2->n1)
    -- Candidato 3:
    n2.(neighbors-n2->n1) = n2.^(neighbors-n2->n1) <=>
    n2.^(neighbors-n2->n1) in n2.(neighbors)
}
```

Figura 8. Predicados CTree con *Holes* y candidatos de reparación para cada uno

Otro punto a trabajar es evaluar fuertemente la generación de candidatos como reparaciones que sólo cumplen con los test brindados pero no solucionan el problema general. El oráculo (test) es muy débil para asegurar las propuestas como reparación [15]. Es por esto que creemos necesaria una evaluación de la técnica que presentamos para evaluar la presencia de falsos positivos en las reparaciones que *ASketch* nos genera.

Referencias

1. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.
2. Renzo Degiovanni, Dalal Alrajeh, Nazareno Aguirre, and Sebastián Uchitel. Automated goal operationalisation based on interpolation and SAT solving. In *36th ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 129–139, 2014.
3. Edsger W. Dijkstra. The humble programmer. 1972.
4. Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. MTSA: the modal transition system analyser. In *23rd IEEE/ACM (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 475–476, 2008.
5. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
6. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
7. Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.
8. Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. The power of "why.ªnd "why not": enriching scenario exploration with provenance. In *Proceedings of the 2017 11th, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 106–116, 2017.
9. Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. Field-exhaustive testing. In *Proceedings of the 24th ACM SIGSOFT, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 908–919, 2016.
10. Germán Regis, César Cornejo, Simón Gutiérrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti, and Marcelo F. Frias. Dynalloy analyzer: a tool for the specification and analysis of alloy models with dynamic behaviour. In *Proceedings of the 2017 11th ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 969–973, 2017.
11. Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. Aunit: A test automation tool for alloy. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 398–403, 2018.
12. Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.
13. Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Mualloy: a mutation testing framework for alloy. In *Proceedings of the 40th ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 29–32, 2018.
14. Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Asketch: a sketching framework for alloy. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 916–919, 2018.
15. Luciano Zemín, Simón Gutiérrez Brida, Ariel Godio, César Cornejo, Renzo Degiovanni, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. An analysis of the suitability of test-based patch acceptance criteria. In *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 14–20, 2017.